เอกสารประกอบการอบรมค่ายโอลิมปิกวิชาการ

สาขาคอมพิวเตอร์

ปีการศึกษา 2564 ค่ายที่ 2

ศูนย์ สอวน. คณะวิทยาศาสตร์และเทคโนโลยี

มหาวิทยาลัยสงขลานครินทร์ วิทยาเขตปัตตานี

21 เมษายน 2565

1. กราฟและชนิดของกราฟ (Graph and its Types)

กราฟทางคณิตศาสตร์แบ่งออกเป็นสองประเภทคือ กราฟแบบไม่มีทิศทาง และกราฟแบบมีทิศทาง

กราฟแบบไม่มีทิศทาง

นิยาม กราฟแบบไม่มีทิศทาง (undirected graph)

กราฟแบบไม่มีทิศทาง $G(V,E)$ ประกอบด้วยเซต V ซึ่งเป็นเซต (ที่ไม่ใช่เซตว่าง) ของ**จุดยอด (vertex/node)** และเซต E ซึ่งเป็นเซตของ**เส้นเชื่อม (edge)**

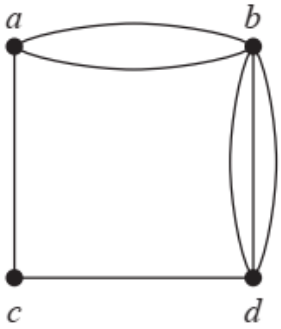
เส้นเชื่อม $e \in E$ แต่ละเส้นสัมพันธ์กับจุดยอด 1 หรือ 2 จุดยอด เรียกจุดยอดเหล่านั้นว่า**จุดปลาย (endpoint)** ของเส้นเชื่อม และกล่าวได้ว่าเส้นเชื่อม**เชื่อม (connect)** จุดปลายของมันไว้ด้วยกัน

ถ้า u และ v เป็นจุดปลายของเส้นเชื่อม เราสามารถเขียน (u,v) หรือ (v,u) แทนเส้นเชื่อม e ได้

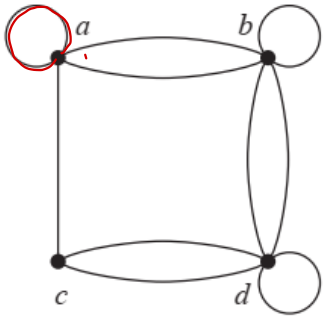
รูปที่ 1 แสดงกราฟ A และ กราฟ B ซึ่งเป็นตัวอย่างของกราฟแบบไม่มีทิศทาง กราฟ A มีเซตของจุดยอดคือ $\{a,b,c,d\}$ และเซตของเส้นเชื่อมคือ $\{(a,b),(a,b),(a,c),(b,d),(b,d),(b,d),(c,d)\}$

เส้นเชื่อมที่มีจุดปลายเป็นจุดยอดเดียวกัน เรียกว่า loop ตัวอย่างของ loop ในกราฟ B คือเส้นเชื่อม (a,a) , (b,b) และ (d,d)

เส้นเชื่อมใดที่มีจุดปลายทั้งสองจุดซ้ำกับเส้นเชื่อมอื่นในกราฟเดียวกัน เรียกว่า multiple edge ตัวอย่างของ **multiple edge** ในกราฟ B คือเส้นเชื่อม $(a,b),(a,b),(b,d),(b,d)(c,d)$ และ (c,d)



กราฟ A



กราฟ B

รูปที่ 1

กราฟแบบไม่มีทิศทางที่ไม่มี loop และไม่มี multiple edge เรียกว่า**กราฟอย่างง่าย (simple graph)**

ถ้า (u,v) เป็นเส้นเชื่อมในกราฟแบบไม่มีทิศทางแล้ว เราเรียกว่า u **adjacent** กับ v และ v **adjacent** กับ u และเรียกว่า u และ v **เป็นเพื่อนบ้านกัน (neighbor)**

เซตของเพื่อนบ้าน (neighborhood) ของจุดยอด u เขียนแทนด้วย $N(u)$ คือเซตของจุดยอดทุกจุดยอดที่ adjacent กับ u

ถ้า $e=(u,v)$ เป็นเส้นเชื่อมในกราฟแบบไม่มีทิศทางแล้ว เราเรียกว่าเส้นเชื่อม e incident กับจุดยอด u และเรียกว่าเส้นเชื่อม e incident กับจุดยอด v และกล่าวว่าเส้นเชื่อม e **เชื่อม (connect)** u กับ v

ดีกรีของจุดยอด u เขียนแทนด้วย $deg(u)$ คือจำนวนเส้นเชื่อมที่ incident กับ u ยกเว้น loop ที่จะนับเป็นสองเท่า

ทฤษฎี 1 Handshaking Theorem

ให้ $G(V,E)$ เป็นกราฟแบบไม่มีทิศทาง แล้ว $2|E|=\sum_{v\in V}\deg(v)$

กราฟแบบมีทิศทาง

นิยาม กราฟแบบมีทิศทาง (directed graph)

กราฟแบบมีทิศทาง $G(V,E)$ ประกอบด้วยเซต V ซึ่งเป็นเซต (ที่ไม่ใช่เซตว่าง) ของ**จุดยอด (vertex/node)** และเซต E ซึ่งเป็นเซตของ**เส้นเชื่อมแบบมีทิศทาง (directed edge)**

เส้นเชื่อม $e\in E$ แต่ละเส้นสัมพันธ์กับคู่อันดับ (ordered pair) (u,v) ของจุดยอด

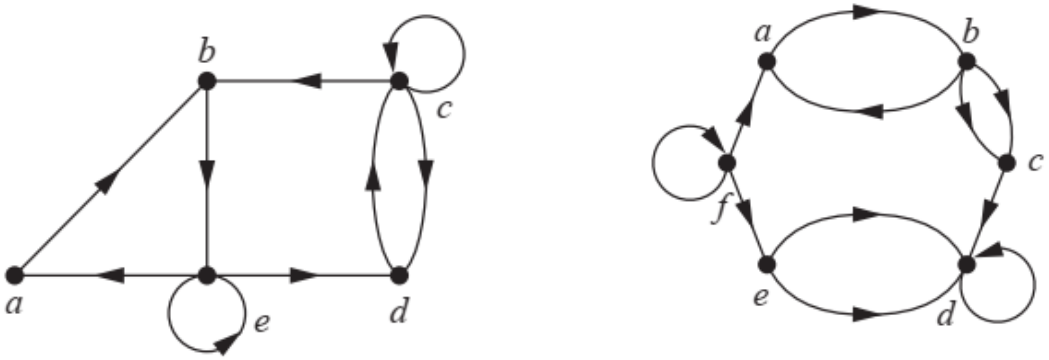
เรียกจุดยอด u ว่า**จุดเริ่ม (start)** ของเส้นเชื่อม และ เรียกเรียกจุดยอด v ว่า**จุดปลาย (end)** ของเส้นเชื่อม

ถ้า u เป็นจุดเริ่มและ v เป็นจุดปลายของเส้นเชื่อม เราสามารถเขียน (u,v) แทนเส้นเชื่อม e ได้

รูปที่ 2 แสดงกราฟ C และ กราฟ D ซึ่งเป็นตัวอย่างของกราฟแบบมีทิศทางกราฟ C มีเซตของจุดยอดคือ $\{a,b,c,d,e\}$ และเซตของเส้นเชื่อมคือ $\{(a,b),(b,e),(c,b),(c,c),(c,d),(d,c),(e,a),(e,d),(e,e)\}$

เส้นเชื่อมที่มีจุดปลายเป็นจุดยอดเดียวกันเรียกว่า **loop** ตัวอย่างของ loop ในกราฟ D คือเส้นเชื่อม (f,f) และ (d,d)

เส้นเชื่อมใดที่มีจุดเริ่มและจุดปลายซ้ำกับจุดเริ่มและจุดปลายของเส้นเชื่อมอื่นในกราฟเดียวกัน เรียกว่า **multiple edge** ตัวอย่างของ **multiple edge** ในกราฟ D คือเส้นเชื่อม $(b,c),(b,c),(e,d)$ และ (e,d) กราฟ C ไม่มี multiple edge



กราฟ C กราฟ D

รูปที่ 2

กราฟแบบมีทิศทางที่ไม่มี loop และไม่มี multiple edge เรียกว่า**กราฟอย่างง่ายแบบมีทิศทาง (directed simple graph)**

ถ้า (u,v) เป็นเส้นเชื่อมในกราฟแบบมีทิศทางแล้ว เราเรียกว่า u **adjacent to** v และ v **adjacent from** u

in-degree ของจุดยอด u เขียนแทนด้วย $deg^-(u)$ คือจำนวนเส้นเชื่อมที่มี u เป็นจุดปลาย

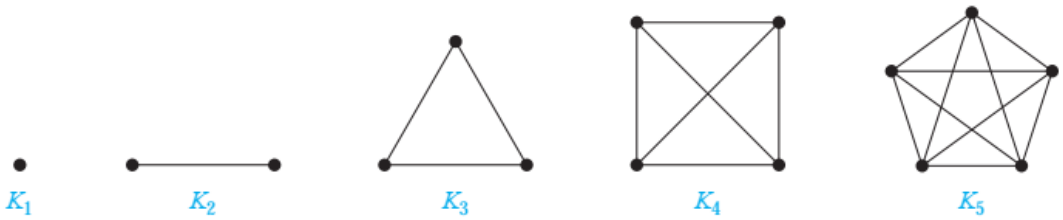
out-degree ของจุดยอด u เขียนแทนด้วย $deg^+(u)$ คือจำนวนเส้นเชื่อมที่มี u เป็นจุดเริ่ม

ทฤษฎี 2

ให้ $G(V,E)$ เป็นกราฟแบบมีทิศทาง แล้ว $\sum_{v\in V}\deg^{-}(v)=\sum_{v\in V}\deg^{+}(v)$

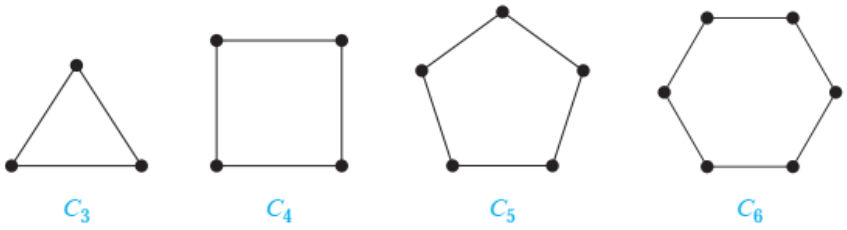
กราฟอย่างง่ายบางรูปแบบ

Complete Graphs ที่มี n จุดยอดเขียนแทนด้วย K_n คือกราฟอย่างง่ายที่มีเส้นเชื่อมระหว่างทุกคู่ของจุดยอดเสมอ รูปที่ 3 แสดงตัวอย่างของ Complete Graphs



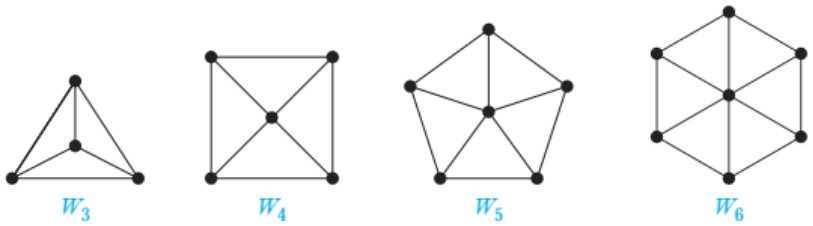
รูปที่ 3

Cycles ที่มี $n\geq 3$ จุดยอดเขียนแทนด้วย C_n คือกราฟอย่างง่ายที่มีเซตของจุดยอด $\{v_1,v_2,...,v_n\}$ และเซตของเส้นเชื่อม $\{(v_1,v_2),(v_2,v_3)...,(v_{n-1},v_n),(v_n,v_1)\}$ รูปที่ 4 แสดงตัวอย่างของ Cycles



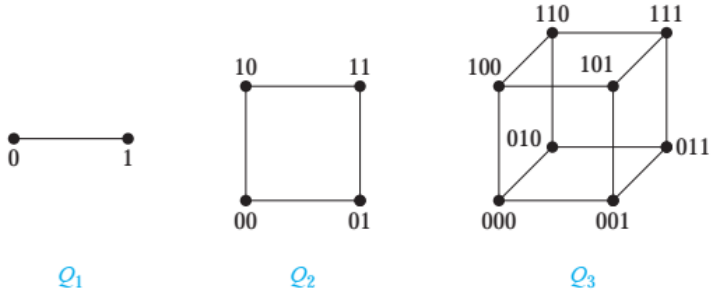
รูปที่ 4

Wheels เขียนแทนด้วย W_n คือกราฟอย่างง่ายที่เกิดจากการเพิ่มจุดยอด 1 จุดให้กับ C_n และเพิ่มเส้นเชื่อม n เส้นเชื่อมจากจุดยอดนี้ไปยังทุกจุดยอดที่เหลือ รูปที่ 5 แสดงตัวอย่างของ Wheels



รูปที่ 5

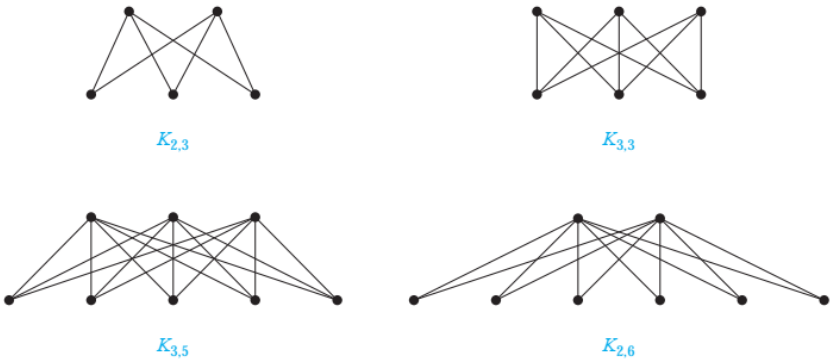
n-Cubes (n-dimensional hypercube) เขียนแทนด้วย Q_n คือกราฟอย่างง่ายที่ใช้บิทตริงความยาว n จำนวน 2^n บิทสตริงแทนจุดยอด และจุดยอดสองจุดยอด incident กันก็ต่อเมื่อบิทตริงที่แทนจุดยอดนั้นต่างกันเพียง 1 ตำแหน่ง รูปที่ 6 แสดงตัวอย่างของ n-Cubes



รูปที่ 6

Bipartite Graphs คือกราฟอย่างง่ายที่เซตของจุดยอด V สามารถถูกแบ่งเป็น 2 เซตคือ V_1 และ V_2 โดยที่ $V_1 \cap V_2 = \varnothing$ และเส้นเชื่อมทุกเส้นต้องมีจุดปลายอยู่ใน V_1 และอีกจุดปลายอยู่ใน V_2 เรียก (V_1, V_2) ว่า bipartite ของเซตของจุดยอด

Complete Bipartite Graphs เขียนแทนด้วย $K_{m,n}$ เป็น Bipartite graph ที่มี bipartite (V_1, V_2) เมื่อ $|V_1| = m$ และ $|V_2| = n$ และมีเส้นเชื่อมระหว่างทุกคู่จุดยอด $u \in V_1$ และ $v \in V_2$ รูปที่ 7 แสดงตัวอย่างของ Complete Bipartite Graphs



รูปที่ 7

2. การแทนกราฟ (Representation of Graph)

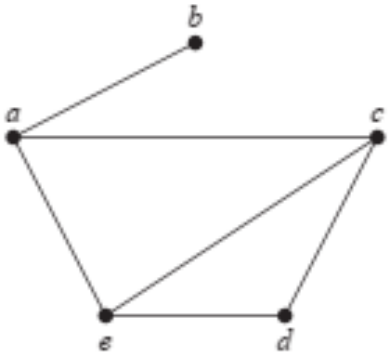
การแทนกราฟในรูปของโครงสร้างข้อมูลสำหรับการเขียนโปรแกรมมี 2 วิธีที่สำคัญคือ การใช้ adjacency list และการใช้ adjacency matrix การเลือกวิธีการแทนกราฟมีความสำคัญมากเพราะ time complexity ของขั้นตอนวิธีที่ทำงานกับกราฟหรือบนกราฟนั้น ขึ้นอยู่กับวิธีการแทนกราฟด้วย บทเรียนนี้จะกล่าวถึงการแทน undirected graph และ directed graph ด้วย adjacency list เป็นหลัก แต่จะมีตัวอย่างโปรแกรมของการแทนกราฟด้วย adjacency matrix ด้วยหนึ่งตัวอย่าง

นอกจากนั้นแล้ว ยังจะมีตัวอย่างของการแทนกราฟแบบมีน้ำหนัก (weighted graph) กราฟแบบมีน้ำหนักคือกราฟที่เส้นเชื่อมมีค่าน้ำหนักที่เป็นตัวเลขกำกับอยู่ กราฟแบบมีน้ำหนักอาจเป็นแบบไม่มีทิศทาง (weighted undirected graph) หรือแบบมีทิศทาง (weighted directed graph) ก็ได้

การแทนกราฟ adjacency list

สำหรับ undirected graph การแทนกราฟด้วย adjacency list คือการให้แต่ละจุดยอด u ในกราฟมีลิสต์ของจุดยอด v ที่ adjacent กับ u (ดูตัวอย่างที่ 1)

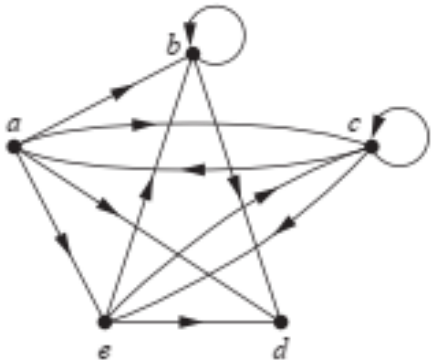
ตัวอย่างที่ 1



Vertex	Adjacency Vertices
a	b, c, e
b	a
c	a, d, e
d	c, e
e	a, c, d

สำหรับ directed graph การแทนกราฟด้วย adjacency list คือการให้แต่ละจุดยอด u ในกราฟมีลิสต์ของจุดยอด v ที่ adjacent from u

ตัวอย่างที่ 2



Vertex	Adjacency Vertices
a	b, c, d, e
b	b, d
c	a, c, e
d	
e	b, c, d

โครงสร้างข้อมูลที่เหมาะสมสำหรับการเก็บ Vertex จะเป็นอาร์เรย์, array หรือ vector ก็ได้ เนื่องจากเรามักจะทราบจำนวนจุดยอดของกราฟเมื่อเริ่มโปรแกรม และโครงสร้างข้อมูลสำหรับเก็บ Adjacency Vertices ของแต่ละจุดยอด จะเป็น vector หรือ linked list ก็ได้ (จำนวนจุดยอดที่ adjacent กับ/ adjacent from จะไม่เกินจำนวนเส้นเชื่อมทั้งหมด)

การเลือกใช้ linked list สำหรับเก็บ Adjacency Vertices ช่วยประหยัดหน่วยความจำให้กรณีที่กราฟมีเส้นเชื่อมน้อย อย่างไรก็ตามการเข้าถึงข้อมูลของ vector ก็เร็วกว่า linked list สำหรับบทเรียนนี้จะเลือกใช้ vector เพราะขั้นตอนวิธี DFS และ BFS สำหรับบทเรียนนี้ จะต้องเข้าถึงจุดยอดที่อยู่ติดกันใน Adjacency Vertices ทำให้ vector ที่ใช้พื้นที่ในหน่วยความจำที่ติดกัน และเก็บทั้งหมดใน cache ได้ มีข้อได้เปรียบในด้านความเร็ว

ทางเลือกอื่นของ Adjacency Vertices คือ set และ unordered_set การเลือก set มีข้อเสียคือ time complexity ของการเพิ่มเส้นเชื่อมเป็น $O(\log N)$ เมื่อ N เป็นจำนวนจุดยอด (เทียบกับ $\theta(1)$ ของ vector) แต่มีข้อดีคือหากต้องมีการตรวจสอบว่ามีเส้นเชื่อมระหว่างจุดยอดคู่ใด ก็สามารถทำได้ใน $O(\log N)$ (เทียบกับ vector $O(M)$ เมื่อ M เป็นจำนวนเส้นเชื่อม) การเลือก unordered_set มีข้อดีคือ time complexity ของการเพิ่มเส้นเชื่อมและการตรวจสอบว่ามีเส้นเชื่อมระหว่างจุดยอดคู่ใดส่วนใหญ่เป็น $\theta(1)$ แต่ทั้ง set และ unordered_set ไม่สามารถจัดการกับ multiple edge ได้ เพราะทั้งสอง container ไม่อนุญาตให้มีสมาชิกที่ซ้ำ

โปรแกรม 2.1 แสดงการแทน undirected graph และ directed graph ด้วย adjacency list โดยการใช้อาร์เรย์เก็บ Vertex และ vector เก็บ Adjacency Vertices

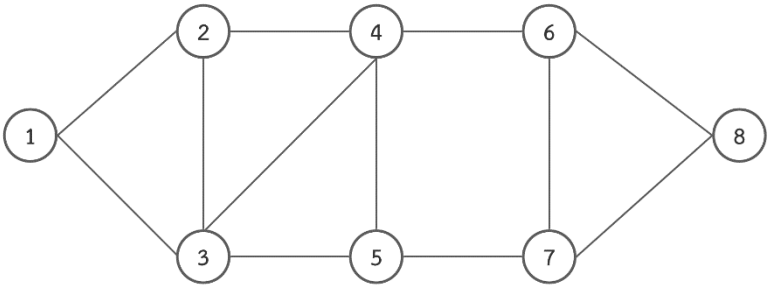
โปรแกรมที่ 2.1 การแทน undirected graph และ directed graph ด้วย adjacency list

```
1.  #include<iostream>
2.  #include<vector>   // vector
3.  using namespace std;
4.
5.  class UGraph{     // Unweighted undirected graph
6.      public:
7.      int V;        // number of nodes
8.      int E;        // number of edges
9.      vector<int> *adj;
10.
11.     // constructor
12.     UGraph(int N){
13.         this->V = N; this->E = 0;
14.         adj = new vector<int> [N+1];
15.     }
16.     // add an undirected edge (u-v) to the graph
17.     void addEdge(int u, int v){
18.         adj[u].emplace_back(v);
19.         adj[v].emplace_back(u);
20.         E++;
21.     }
22.     // display adjacency list of the graph
23.     void printGraph() const{
24.         cout << "Undirect::Node="<< V << " Edge=" << E << "\n";
25.         for(int i=1; i<=V; i++){
26.             cout << i << ": ";
27.             for(auto &u : adj[i])
28.                 cout << u << " ";
29.             cout << "\n";
30.         }
31.     }
32. };
33. class DiGraph{     // Unweighted directed graph
34.     public:
35.     int V;        // number of nodes
36.     int E;        // number of edges
37.
38.     vector<int> *adj;
39.     // constructor
40.     DiGraph(int N){
41.         this->V = N; this->E = 0;
42.         adj = new vector<int> [N+1];
43.     }
```

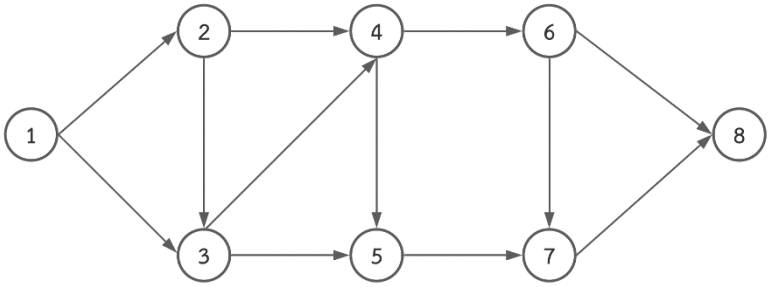
```
44.       // add an directed edge (u->v) to the graph
45.       void addEdge(int u, int v){
46.           adj[u].emplace_back(v);
47.           E++;
48.       }
49.       // display adjacency list of the graph
50.       void printGraph() const{
51.           cout << "Direct::Node="<< V << " Edge=" << E << "\n";
52.           for(int i=1; i<=V; i++){
53.               cout << i << ": ";
54.               for(auto &u : adj[i])
55.                   cout << u << " ";
56.               cout << "\n";
57.           }
58.       }
59.   };
60.
61.   int main(){
62.       int N = 8;
63.       UGraph G1(N);
64.       int endNode1[] = {1,1,2,2,3,3,4,4,5,6,6,7};
65.       int endNode2[] = {2,3,3,4,4,5,5,6,7,7,8,8};
66.       for(int i=0; i<12; i++)
67.           G1.addEdge(endNode1[i],endNode2[i]);
68.       G1.printGraph();
69.
70.       DiGraph G2(N);
71.       int startNode[] = {1,1,2,2,3,3,4,4,5,6,6,7};
72.       int endNode[] = {2,3,3,4,4,5,5,6,7,7,8,8};
73.       for(int i=0; i<12; i++)
74.           G2.addEdge(startNode[i],endNode[i]);
75.       G2.printGraph();
75.
76.       return 0;
77.   }
78.
79.
80.
81.
```

ผลลัพธ์

```
Undirect::Node=8 Edge=12
1: 2 3
2: 1 3 4
3: 1 2 4 5
4: 2 3 5 6
5: 3 4 7
6: 4 7 8
7: 5 6 8
8: 6 7
```




```
Direct::Node=8 Edge=12
1: 2 3
2: 3 4
3: 4 5
4: 5 6
5: 7
6: 7 8
7: 8
8:
```



ในกรณีกราฟเป็นกราฟแบบมีน้ำหนัก เราสามารถปรับ vector ที่ใช้สำหรับ Adjacency Vertices ให้เก็บค่า 2 ค่าคือ (1) จุดยอดที่ adjacent กับ/adjacent from (2) ค่าน้ำหนักของเส้นเชื่อม โดยการใช้ pair หรือจะเขียนเป็น struct/class ก็ได้

โปรแกรม 2.2 แสดงการแทน weighed undirected graph และ weighted directed graph ด้วย adjacency list โดยการใช้อาร์เรย์เก็บ Vertex และ vector เก็บ Adjacency Vertices

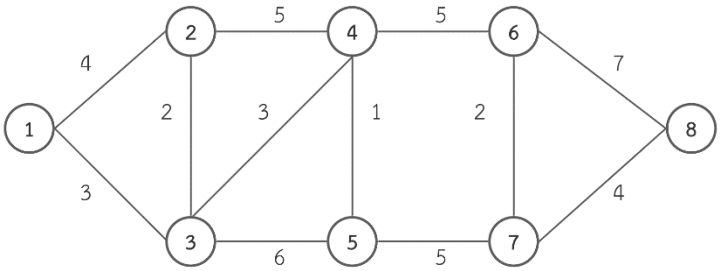
โปรแกรมที่ 2.2 การแทน weighted undirected graph และ weighted directed graph ด้วย adjacency list

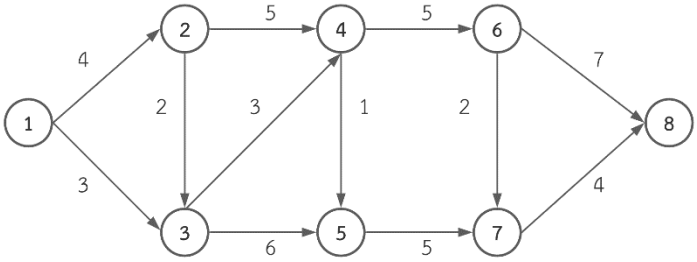
```
1.  #include<iostream>
2.  #include<vector>   // vector
3.  #include<utility>  // pair
4.  using namespace std;
5.
6.  class UGraph{     // Weighted undirected graph
7.      public:
8.      int V;        // number of nodes
9.      int E;        // number of edges
10.     vector< pair <int, int > > *adj;
11.     // constructor
12.     UGraph(int N){
13.         this->V = N; this->E = 0;
14.         adj = new vector< pair <int, int > > [N+1];
15.     }
16.     // add an undirected edge (u-v) with weight to the graph
17.     void addEdge(int u, int v, int w){
18.         adj[u].emplace_back(make_pair(v,w));
19.         adj[v].emplace_back(make_pair(u,w));
20.         E++;
21.     }
22.     // display adjacency list of the graph
23.     void printGraph() const{
24.         cout << "Undirect::Node="<< V << " Edge=" << E << "\n";
25.         for(int i=1; i<=V; i++){
26.             cout << i << ": ";
27.             for(auto &u : adj[i]){
28.                 cout << "(" << u.first << "," << u.second << ") ";
29.             }
30.             cout << "\n";
31.         }
32.     }
33. };
34.
35.
36. class DiGraph{     // Unweighted directed graph
37.     public:
38.     int V;        // number of nodes
39.     int E;        // number of edges
40.     vector< pair <int, int > > *adj;
41.     // constructor
42.     DiGraph(int N){
43.         this->V = N; this->E = 0;
44.         adj = new vector< pair <int, int > > [N+1];
45.     }
```

```
46.       // add an directed edge (u->v) with weight w to the graph
47.       void addEdge(int u, int v, int w){
48.           adj[u].emplace_back(make_pair(v,w));
49.           E++;
50.       }
51.       // display adjacency list of the graph
52.       void printGraph() const{
53.           cout << "Direct::Node="<< V << " Edge=" << E << "\n";
54.           for(int i=1; i<=V; i++){
55.               cout << i << ": ";
56.               for(auto &u : adj[i])
57.                   cout << "(" << u.first << "," << u.second << ") ";
58.               cout << "\n";
59.           }
60.       }
61.   };
62.
63.   int main(){
64.       int N = 8;
65.       UGraph G1(N);
66.       int endNode1[] = {1,1,2,2,3,3,4,4,5,6,6,7};
67.       int endNode2[] = {2,3,3,4,4,5,5,6,7,7,8,8};
68.       int weight[] = {4,3,2,5,3,6,1,5,5,2,7,4};
69.       for(int i=0; i<12; i++)
70.           G1.addEdge(endNode1[i],endNode2[i],weight[i]);
71.       G1.printGraph();
72.
73.       DiGraph G2(N);
74.       int startNode[] = {1,1,2,2,3,3,4,4,5,6,6,7};
75.       int endNode[] = {2,3,3,4,4,5,5,6,7,7,8,8};
75.       for(int i=0; i<12; i++)
76.           G2.addEdge(startNode[i],endNode[i],weight[i]);
77.       G2.printGraph();
78.
79.       return 0;
80.   }
```

ผลลัพธ์

```
Undirect::Node=8 Edge=12
1: (2,4) (3,3)
2: (1,4) (3,2) (4,5)
3: (1,3) (2,2) (4,3)
(5,6)
4: (2,5) (3,3) (5,1)
(6,5)
5: (3,6) (4,1) (7,5)
6: (4,5) (7,2) (8,7)
7: (5,5) (6,2) (8,4)
8: (6,7) (7,4)
Direct::Node=8 Edge=12
1: (2,4) (3,3)
2: (3,2) (4,5)
3: (4,3) (5,6)
4: (5,1) (6,5)
5: (7,5)
6: (7,2) (8,7)
7: (8,4)
8:
```





การแทนกราฟ adjacency matrix

การแทนกราฟ adjacency matrix จะใช้อาร์เรย์ 2 มิติขนาด $N \times N$ เมื่อ $O(N)$ เป็นจำนวนจุดยอดในกราฟ

สำหรับ **unweighted undirected graph** ที่แถวที่ r และคอลัมน์ที่ c และ ที่แถวที่ c และคอลัมน์ที่ r ของอาร์เรย์นี้จะเก็บค่าความจริง (0/1 หรือ true/false) ที่แทนว่าจุดยอด r และ c เป็นเพื่อนบ้านกันหรือไม่ สำหรับ **unweighted directed graph** ที่แถวที่ r และคอลัมน์ที่ c อาร์เรย์นี้จะเก็บค่าความจริง (0/1 หรือ true/false) ที่แทนว่ามีเส้นเชื่อมที่มีจุดเริ่มเป็น r และจุดปลายเป็น c หรือไม่

สำหรับ **weighted undirected graph** ที่แถวที่ r และคอลัมน์ที่ c และ ที่แถวที่ c และคอลัมน์ที่ r ของอาร์เรย์นี้จะเก็บค่าน้ำหนักของเส้นเชื่อม (r,c) สำหรับ **weighted directed graph** ที่แถวที่ r และคอลัมน์ที่ c ของอาร์เรย์นี้จะเก็บค่าน้ำหนักของเส้นเชื่อม (r,c)

โปรแกรม 2.3 แสดงการแทน unweighted undirected graph ด้วย adjacency matrix

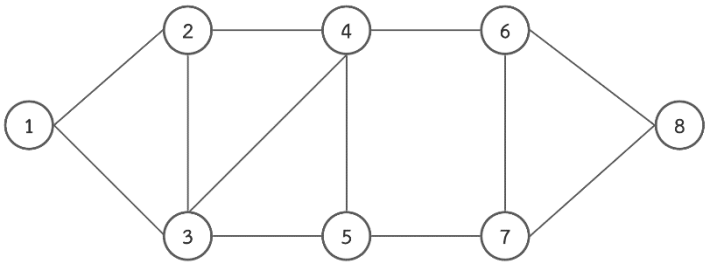
โปรแกรมที่ 2.3 การแทน unweighted undirected graph ด้วย adjacency matrix

```
1.  #include<iostream>
2.  #include<vector>   // vector
3.  using namespace std;
4.  class UGraph{     // Unweighted undirected graph
5.      public:
6.      int V;        // number of nodes
7.      int E;        // number of edges
8.      vector< vector<int> > mat;
9.
10.     // constructor
11.     UGraph(int N){
12.         this->V = N; this->E = 0;
13.         mat.resize(N+1);
14.         for(auto &row: mat)
15.             row.resize(N+1);
16.     }
17.     // add an undirected edge (u-v) to the graph
18.     void addEdge(int u, int v){
19.         (mat[u])[v] = 1;
20.         (mat[v])[u] = 1;
21.         E++;
22.     }
23.     // display adjacency matrix of the graph
24.     void printGraph() const{
25.         cout << "Undirect::Node="<< V << " Edge=" << E << "\n  ";
26.         for(int i=1; i<=V; i++)
27.             cout << i << " ";
28.         cout << "\n";
29.         for(int r=1; r<=V; r++){
30.             cout << r << " ";
31.             for(int c=1; c<=V; c++)
32.                 cout << (mat[r])[c]  << " ";
33.             cout << "\n";
34.         }
35.     }
36. };
37.
38.
39.
```

```
40.  int main(){
41.      int N = 8;
42.      UGraph G1(N);
43.      int endNode1[] = {1,1,2,2,3,3,4,4,5,6,6,7};
44.      int endNode2[] = {2,3,3,4,4,5,5,6,7,7,8,8};
45.      for(int i=0; i<12; i++)
46.          G1.addEdge(endNode1[i],endNode2[i]);
47.      G1.printGraph();
48.      return 0;
49.  }
```

ผลลัพธ์

```
Undirect::Node=8 Edge=12
  1 2 3 4 5 6 7 8
1 0 1 1 0 0 0 0 0
2 1 0 1 1 0 0 0 0
3 1 1 0 1 1 0 0 0
4 0 1 1 0 1 1 0 0
5 0 0 1 1 0 0 1 0
6 0 0 0 1 0 0 1 1
7 0 0 0 0 1 1 0 1
8 0 0 0 0 0 1 1 0
```

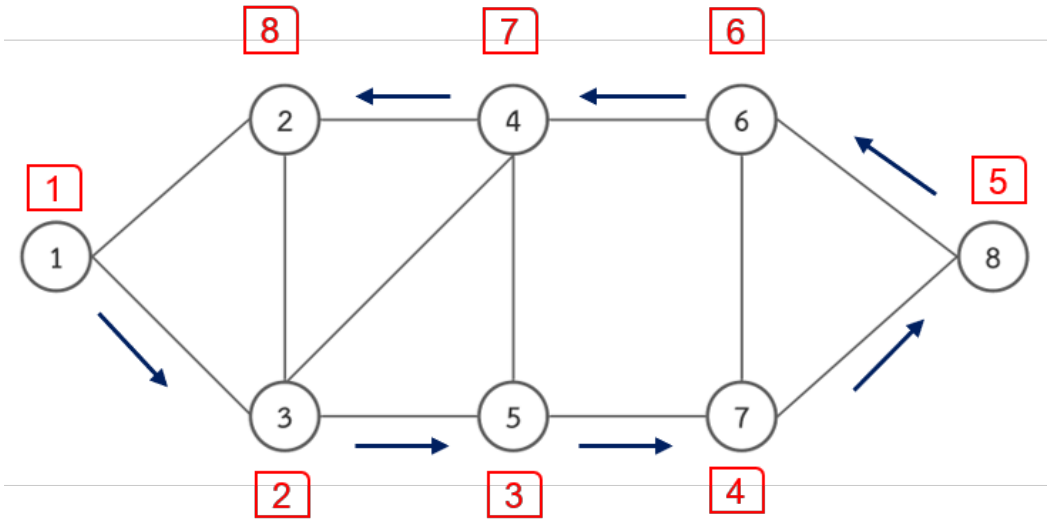


3. Depth First Search on Graph

ทำงานของขั้นตอนวิธี Depth first search (DFS) บนกราฟมีข้อแตกต่างเพียงเล็กน้อย เมื่อเทียบกับการทำงานบนโครงสร้างข้อมูลรูปแบบ 2 มิติ (อาร์เรย์ 2 มิติ) ดังต่อไปนี้

- DFS บนอาร์เรย์ 2 มิติเป็นการ visit cell ในอาร์เรย์ แต่ DFS บนกราฟเป็นการ visit จุดยอดในกราฟ
- การระบุ cell ในอาร์เรย์ 2 มิติใช้ค่าสองค่าคือเลขกำกับแถวและเลขกำกับหลัก แต่การระบุจุดยอดในกราฟใช้หนึ่งค่าคือเลขกำกับจุดยอด (ชื่อของจุดยอด)
- ทิศทางในการท่องอาร์เรย์ 2 มิติของ DFS คือทิศทางรอบ ๆ cell ที่อยู่ปัจจุบัน (ซึ่งมีได้ถึง 8 ทิศ) แต่ทิศทางในการท่องกราฟถูกกำหนดโดยจุดยอดที่ adjacent (กรณี undirected graph) กับหรือ adjacent from (กรณี directed graph) จุดยอดที่อยู่ปัจจุบัน
- การท่องอาร์เรย์ 2 มิติของ DFS ต้องตรวจสอบการออกนอกพื้นที่ของอาร์เรย์ (out of bound) ซึ่งไม่จำเป็นสำหรับการท่องกราฟ

รูปที่ 8 แสดง ลำดับในการ visit จุดยอด (Order of visit) ของ DFS (เรียงตามลำดับการ pop จาก stack)



รูปที่ 8

โปรแกรมที่ 3.1 การทำงานของ DFS บน unweighted undirected graph

```
1.  #include<iostream>
2.  #include<vector>   // vector
3.  #include<stack>    // stack
4.  #define MAX 2005
5.  using namespace std;
6.  class UGraph{     // Unweighted undirected graph
7.      public:
8.      int V;        // number of nodes
9.      int E;        // number of edges
10.     vector<int> *adj;
11.     // default constructor
12.     UGraph(){
13.         this->V = 0; this->E = 0;
14.     }
15.     // constructor
16.     UGraph(int N){
17.         this->V = N; this->E = 0;
18.         adj = new vector<int> [N+1];
19.     }
20.     // reset graph to size N with no edge
21.     void reset(int N){
22.         this->V = N; this->E = 0;
23.         adj = new vector<int> [N+1];
24.     }
25.     // add an undirected edge (u-v) to the graph
26.     void addEdge(int u, int v){
27.         adj[u].emplace_back(v);
28.         adj[v].emplace_back(u);
29.         E++;
30.     }
31.     // display adjacency list of the graph
32.     void printGraph() const{
33.         cout << "Undirect::Node="<< V << " Edge=" << E << "\n";
34.         for(int i=1; i<=V; i++){
35.             cout << i << ": ";
36.             for(auto &u : adj[i])
37.                 cout << u << " ";
38.             cout << "\n";
39.         }
40.     }
41. };
42.
43.
44.
45.
46.
47.
```

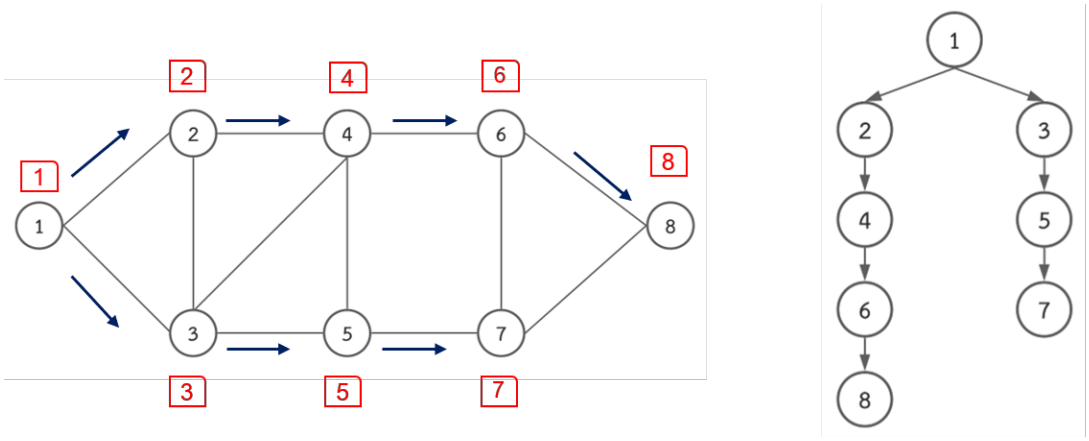
```
48.  // Globals
49.  UGraph G;              // the graph
50.  int N,M;               // number of nodes and edges
51.  bool visit[MAX+1];     // keeps tracks of visited nodes
52.  int parent[MAX+1];     // keep track of parents
53.  stack<int> S;          // for DFS
54.
55.  // Forwards
56.  void iterativeDFS(UGraph &G, int s); // Iterative DFS
57.  void printParent();    // print parent of DFS tree
58.
59.  int main(){
60.      ios_base::sync_with_stdio(false);   // avoid syn C++ streams
61.      cin.tie(NULL);                      // flood cout before cin
62.
63.      cin >> N >> M;
64.      G.reset(N);
65.      for(int i=0; i <M; i++){
66.          int u,v;
67.          cin >> u >> v;
68.          G.addEdge(u,v);
69.      }
70.
71.      int start = 1;
72.      iterativeDFS(G,start);
73.
74.      return 0;
75.  }
75.  void iterativeDFS(UGraph &G, int s){
76.      // push just visited node to stack
77.      // mark node as visited
78.      S.push(s);
79.      // iterate until stack is empty
80.      while(!S.empty()){
81.          int u = S.top();
82.          S.pop();
83.          if(!visit[u]){
84.              visit[u] = true;   // mark visit when pop
85.              cout << u << " ";
86.          }
87.          for(auto &v: G.adj[u]){
88.              if(!visit[v]){
89.                  S.push(v);
90.              }
91.          }
92.      }
93.  }
94.
```

ผลลัพธ์

Input	Output
8 12 **1 2** **1 3** **2 3** **2 4** **3 4** **3 5** **4 5** **4 6** **5 7** **6 7** **6 8** **7 8**	1 3 5 7 8 6 4 2

4. Breath First Search on Graph

การทำงานของ Breath First Search (BFS) บนกราฟจะให้ลำดับในการ visit และ BFS Tree ที่แตกต่างจาก DFS รูปที่ 9 แสดง (1) ลำดับในการ visit จุดยอด (Order of visit) ซึ่งเรียงตามลำดับการ **push** เข้าใน **queue** และ (2) BFS Tree (ซึ่งแสดงความสัมพันธ์ parent-child ระหว่างจุดยอด) จะเห็นได้ว่าลำดับในการ visit จุดยอดเป็นลำดับเดียว level-order visit ของ BFS Tree นอกจากนั้นแล้ว BFS Tree ยังแสดงระยะทาง (distance) จากจุดเริ่มมายังทุกจุดยอด เมื่อใช้ BFS ในการท่องกราฟ ตัวอย่างเช่นจุดยอด 6 มีระยะทางเท่ากับ 3 จากจุดเริ่ม (จุดยอด 1) สำหรับ unweighted graph แล้วระยะทางที่ได้จาก BSF Tree เป็นระยะทางที่สั้นที่สุดจากจุดเริ่มมายังจุดยอดนั้น ๆ (ในกรณี weighed graph ต้องใช้ขั้นตอนวิธีอื่นในการหาระยะทางที่สั้นที่สุดจากจุดเริ่มมายังจุดยอดอื่น ๆ ในกราฟ)



BFS

รูปที่ 9

โปรแกรมที่ 3.2 แสดงการทำงานของ BFS บน unweighted undirected graph โปรแกรมแสดงแสดงความสัมพันธ์ parent-child ของจุดยอด และระยะทางจากจุดเริ่ม (จุดยอด 1) มายังจุดยอดอื่น ๆ ในกราฟ

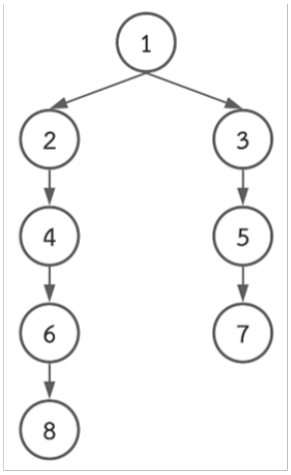
โปรแกรมที่ 3.2 การทำงานของ BFS บน unweighted undirected graph

```
1.    #include<iostream>
2.    #include<vector>   // vector
3.    #include<queue>    // queue
4.    #define MAX 2005
5.    using namespace std;
6.    class UGraph{     // Unweighted undirected graph
7.        public:
8.        int V;        // number of nodes
9.        int E;        // number of edges
10.       vector<int> *adj;
11.       // default constructor
12.       UGraph(){
13.           this->V = 0; this->E = 0;
14.       }
15.       // constructor
16.       UGraph(int N){
17.           this->V = N; this->E = 0;
18.           adj = new vector<int> [N+1];
19.       }
20.       // reset graph to size N with no edge
21.       void reset(int N){
22.           this->V = N; this->E = 0;
23.           adj = new vector<int> [N+1];
24.       }
25.       // add an undirected edge (u-v) to the graph
26.       void addEdge(int u, int v){
27.           adj[u].emplace_back(v);
28.           adj[v].emplace_back(u);
29.           E++;
30.       }
31.       // display adjacency list of the graph
32.       void printGraph() const{
33.           cout << "Undirect::Node="<< V << " Edge=" << E << "\n";
34.           for(int i=1; i<=V; i++){
35.               cout << i << ": ";
36.               for(auto &u : adj[i])
37.                   cout << u << " ";
38.               cout << "\n";
39.           }
40.       }
41.   };
42.   // Globals
43.   UGraph G;             // the graph
44.   int N,M;              // number of nodes and edges
45.   bool visit[MAX+1];    // keeps tracks of visited nodes
```

```
46.   int parent[MAX+1];      // keep track of parents
47.   int dist[MAX+1];        // distance from start to nodes
48.   queue<int> Q;           // for BFS
49.
50.   // Forwards
51.   void iterativeBFS(UGraph G, int s);   // Iterative BFS
52.   void printParent();     // print parent of DFS tree
53.   void printDistance();   // print distance
54.   int main(){
55.       cin >> N >> M;
56.       G.reset(N);
57.       for(int i=0; i <M; i++){
58.           int u,v;
59.           cin >> u >> v;
60.           G.addEdge(u,v);
61.       }
62.
63.       int start = 1;
64.       iterativeBFS(G,start);
65.       printParent();
66.       printDistance();
67.
68.       return 0;
69.   }
70.   void iterativeBFS(UGraph G, int s){
71.       // push just visited node to queue
72.       // mark node as visited
73.       Q.push(s);
74.       visit[s] = true;  // mark visit when push
75.       cout << s << " ";
75.       dist[s] = 0;
76.       parent[s] = s;
77.
78.       // iterate until queue is empty
79.       while(!Q.empty()){
80.           int u = Q.front();
81.           Q.pop();
82.
83.           for(auto &v: G.adj[u]){
84.               if(!visit[v]){
85.                   Q.push(v);
86.                   cout << v << " ";
87.                   visit[v] = true;  // mark visit when push
88.                   parent[v] = u;
89.                   dist[v] = dist[u] + 1;
90.               }
91.           }
92.       }
93.       cout << "\n";
94.   }
```

```
95.   void printParent(){
96.       cout << "Parent:\np[u] => ";
97.       for(int i=1; i<=N; i++)
98.           cout << parent[i] << " ";
99.       cout << "\nu    => ";
100.      for(int i=1; i<=N; i++)
101.          cout << i << " ";
102.      cout << "\n";
103.  }
104.  void printDistance(){
105.      cout << "Distance:\nu =>    ";
106.      for(int i=1; i<=N; i++)
107.          cout << i << " ";
108.      cout << "\ndist => ";
109.      for(int i=1; i<=N; i++)
110.          cout << dist[i] << " ";
111.      cout << "\n";
112.  }
```

ผลลัพธ์

Input	Output
8 12 **1 2** **1 3** **2 3** **2 4** **3 4** **3 5** **4 5** **4 6** **5 7** **6 7** **6 8** **7 8**	1 2 3 4 5 6 7 8 Parent: p[u] => 0 1 1 2 3 4 5 6 u => 1 2 3 4 5 6 7 8 Distance: u => 1 2 3 4 5 6 7 8 dist => 0 1 1 2 2 3 3 4  BFS Tree

อ้างอิง

Kenneth H. Rosen, 2012: **Discrete Mathematics and its Applications 7th ed.**, McGraw-Hill Companies Inc., New York.